EECS-4411M: Test ~~#1~~ #2

"Query Processing"

Electrical Engineering & Computer Science

Lassonde School of Engineering

York University

Family Name:	
Given Name:	
Student#:	
EECS Account:	

Instructor:	Parke Godfrey
Exam Duration:	75 minutes
Term:	Winter 2017

Instructions

- **rules**
  - The test is closed-note, closed-book. Use of a calculator is permitted.
- **answers**
  - Should you feel a question needs an assumption to be able to answer it, write the assumptions you need along with your answer.
  - If you need more room to write an answer, indicate where you are continuing the answer.
  - For multiple choice questions, choose *one* best answer for each of the following. There is no negative penalty for a wrong answer.
- **notation & assumptions**
  - For questions about indexes, assume that the indexes are *dense*.
  - For physical storage of records, assume row store.
- **points**
  - The number of points a given question is worth is marked. (It is worth one point, if not marked.)
  - There are five major parts worth 10 points each, for 50 points in total.

Marking Box	
1.	/10
2.	/10
3.	/10
4.	/10
5.	/10
Total	/50

1. [10pt] **Access Paths.** *Show me the way!* EXERCISE

   Consider table **T**. **T** has 10 million records on 200,000 pages (so, 50 records per page, on average). **T** has columns A. B, C, and D. There are two tree indexes on **T**:
   #1: on B, C
   3 index-pages deep
   100 data entries per page, on average
   #2: on C, D
   3 index-pages deep
   100 data entries per page, on average

   The indexes are *indirect.* (That is, the data records themselves are elsewhere.)

   The number of distinct values of **T**.B is one million, 0, . . . 999999.

   Likewise, the number of distinct values of **T**.C is one million, 0, . . . 999999.

   Consider the parameterized SQL query

```
select *
from ~~R~~ T
where B between :u and :v
  and C between :x and :y
```

   The values for :u, :v, :x, and :y are filled in when the query is called. Assume the predicate between is inclusive of the two values.

   For each of the following questions, answer which is the best *access path* and show the calculation of its I/O cost.[1]

For Questions 1a, 1b, and 1c, assume that index #1 is *unclustered*, but that index #2 is *clustered.*

---

a. [2pt] Assume values
   :u = 491830, :v = 491839
   :x = 100001, :y = 600000

   RF (reduction factor) of range on B: 10/1M = 1/100K; so 10 values, matching 100 records.
   RF of range on C: 500K/1M = 1/2; matches 500K values, 5M records.
   Use index #1: 3 IP (index pages) + 2 DE (data entry pages) + 100 DR (data record pages) = 105 I/O's. (It is an I/O per record, since Index #1 is unclustered.)
   Note that since Index #1's key is B, C, we can check the C condition in the DE's before *fetching the records. We expect only to match* $10/1M * 1/2 * 10M = 50$ *records. In that case, using Index #1: 3 IP + 2 DE + 50 DR (data record pages) = 55 I/O's.*

b. [2pt] Assume values
   :u = 210317, :v = 310316
   :x = 743110, :y = 843109

   RF on range B: 100K/1M = 1/10; so 100K values, 1M records.
   RF on range C: 100K/1M = 1/10; so 100K values, 1M records.
   Using Index #2: 3 IP + 10K DE + 20K DR ≈ 30K I/O's. (Check range on B on the fly.)

[1]For scoring Questions 1a through 1a, it is +1pt for correct *reduction factor*, and +1pt for correct access path and cost estimation.

c. [2pt] Assume the query's select-clause is "select B, C" instead. Assume values
:u = 210317, :v = 510316
:x = 443110, :y = 743109

RF on range B: 300K/1M = 3/10; so 300K values, 3M records.
RF on range C: 100K/1M = 3/10; so 300K values, 3M records.
Using Index #1 as index only*: 3 IP + 30K DE ≈ 30K I/O's. (Check range on C on the fly against DE's.)*

For Questions 1d and 1e, assume that indexes #1 and #2 are both *unclustered*. (Also, assume again the original query with "select * ...".)

---

d. [2pt] Assume values
:u = 901280, :v = 941279
:x = 408339, :y = 428338

RF on range B: 40K/1M = 4/100; so 40K values, 400k records.
RF on range C: 20K/1M = 2/100; so 20K values, 200k records.
Selectivity together is $4/100 * 2/100 * 10M = 8000$*. Using Index #1: 3 IP + 4,000 DE + 8,000 DR ≈ 12,000 I/O's. (Picks up DE pages in Index #1 matching range B; then checks range C in the data entry, fetching page by page—unclustered!—the 8,000 records that match.)*
I'd forgot index #1 covered C in its key here; if it didn't, neither index access path would be better than just the 200K I/O filescan.

---

e. [2pt] Assume values[2]
:u = 841280, :v = 841279
:x = 318339, :y = 418338

Find an access path costing 10,000 I/O's, at most.

RF on range B: 2/1M; so 2 values, 20 records.
RF on range C: 100K/1M = 1/10; so 100K values, 1M records.
Using Index #1: 3 IP + 1 DE + 10 DR = 14 I/O's.
Again, we could check the range C against the DE's on the fly. Then we would need to only fetch $1/10 * 20 = 2$ *records: 3 IP + 1 DE + 2 DR = 6 I/O's.*
If I had put :v = 941279*, then what? Consider* index intersection...

[2] I had meant :v = 941279, but made a typo!

2. [10pt] **Join Algorithms.** *Come join the Join Club!* Analysis

Congratulations! You have gone to work for *Query Forward*, a new Toronto company committed to building the world's fastest relational database system. Your boss is the infamous database researcher Dr. Mark Dogfurry.

For Questions 2a & 2b:
He claims that there is a way to improve *index-nested loop join* (INLJ). He says that one should *always* sort the outer first by the *join key*—that is, the join-condition's attributes—before doing the join, and that this will *reduce* the overall I/O cost.

a. [2pt] Explain a specific scenario in which Dr. Dogfurry's "improvement" to INLJ—to sort the outer first by the join key—*raises* the cost overall of the INLJ operation.
What is the additional cost in your scenario?

Say for the outer that each record matches just one inner record—for instance, a child table joined w/ a parent table over a foreign key, like **Student** *and* **Enrol***—then this did* nothing *to help. (And this is the same if we reverse parent to outer and child to inner. Why?)*
INLJ costs the same as before, but paid additionally *to sort the outer.*

b. [3pt] Explain a specific scenario in which Dr. Dogfurry's "improvement" to INLJ—to sort the outer first by the join key—*lowers* the cost overall of the INLJ operation.
What is the saved cost in your scenario?

Say we have a many-many join (unlike the case in our answer to Question 2a). So there can be repeated *values on the outer,* and *each probe can have multiple records matching on the inner. By having the outer sorted, each subsequent probe by a* repeated *outer value hits the* same *records on its inner probe as the previous probe. And it is highly likely* all *this is still in the buffer pool! Big savings.*

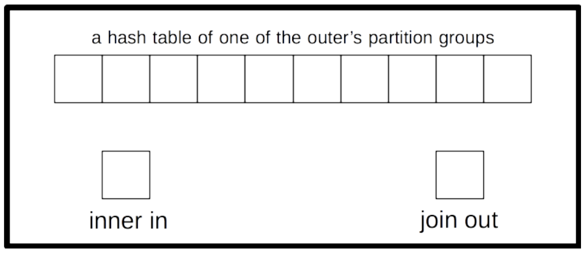
c. [3pt] Consider the join of **R**(<u>A</u>, B) and **S**(<u>B</u>, C) on B—returning A, B, and C—for which A is the primary key of **R**, B the primary key of **S**, and **R** has a foreign key referencing **S** on B.

Consider there is an index for **S** on **B** which is *indirect* (the records are elsewhere), and an index-nested loop join of **R** as the outer and **S** as the inner using the index.

How much less expensive might the join be if the index is clustered versus if it is unclustered?

There is no *difference. Each probe matches* one *record on the inner. This one data-record page is fetched to get the record, for clustered or unclustered.*
In other words, there are no "clustered" hits to take advantage of.

d. [2pt] Draw a sketch of how the buffer pool is used during the *second pass* of a hash join.



3. [10pt] **External Sort.** *Sorting it all out.* Exercise

The standard external sort algorithm works as follows. Assume there are B buffer frames allocated to the task. Let the file to be sorted be R pages.

Pass 0 (the "block" pass):

```
r := 0 // run number
while pages left in file
    e := min(B, how many pages left in file)
    read in e pages
    sort across the e pages
    write out sorted run of length e labelled ⟨0, r⟩
    r++
```

Pass i (for $i > 0$, the "merge" passes):

```
// let k be the number of runs made in pass i - 1
r := 0 // run number
u := 0 // u for last "unused" run from pass i - 1
while u < k
    e := min(u + (B - 1) - 1, k - 1)
    merge runs ⟨i−1, u⟩, ..., ⟨i−1, e⟩ // merge operation
    writing out new run labelled ⟨i, r⟩
    r++
```

Your colleague at *Query Forward*, Dr. Tabitha Récorde, says that she thinks that she can improve on this algorithm. She points out that the last merge operation done in a merge pass may merge much fewer than $B - 1$ runs, which seems wasteful.

For her version then, **Pass 0** remains the same, except now the runs are just labelled sequentially as they are made as $\langle 0\rangle$, $\langle 2\rangle$, $\langle 3\rangle$, ... (There no longer is a pass number.) Then she replaces *all* of the merge passes with her **Merge Phase**.

Merge Phase:

```
r := k // run number; k is #runs from Pass 0
u := 0 // u for last "unused" run
while r - u > 1
    e := min(u + (B - 1) - 1, r - 1)
    merge runs ⟨u⟩, ..., ⟨e⟩ // merge operation
    writing out new run labelled ⟨r⟩
    u = e + 1
    r++
```

---

a. [4pt] Is Dr. Récorde's version an improvement? That is, can her version save I/O's over the standard version? (Perhaps answer Question 3b before this for insight.)
Explain convincingly.

This can effectively reduce a "pass" sometimes with respect to the original approach. And when it does, it is a big win, as it is in this example.
The extra cost is re-reading extra runs to fill out merges; but this can be less than reading "all" in and writing it out again in an extra pass.

b. [4pt] Let the table file **R** consist of 95 pages (with 100 records per page, on average). Trace Dr. Récorde's external sort over **R**. Assume an allocation of five (5) buffer frames for your operation.

State how long each run ⟨i⟩ is. (If many runs are the same, just say, for example, "runs ⟨3⟩ to ⟨9⟩ are 13 pages long each.")

State the overall I/O cost of the external sort.

- *Pass 0: 19 times*
  - *reads in 5 pages, sorts across them*
  - *writes a 5-page run*

  Makes runs $r_0, \ldots, r_{18}$.
  Cost: $2 \times 95 = 190$.
- *Four merges that read in runs $r_0, \ldots, r_{15}$ in sequential groups of four. Writes out four 20-page runs $r_{19}, \ldots, r_{22}$.*
  Cost: $2 \times 80 = 160$.
- *Merge the four runs $r_{16}, \ldots, r_{19}$ to make run r_{23}. Runs $r_{16}, \ldots, r_{18}$ are of length 5 pages; run r_{19} is of length 20 pages.*
  Cost: $2 \times 35 = 70$.
- *Merge the four runs $r_{20}, \ldots, r_{23}$ to make the final run r_{24}. Runs $r_{20}, \ldots, r_{22}$ are of length 20 pages; run r_{23} is of length 35 pages.*
  Cost: $2 \times 95 = 190$.

Run r_{24} is the sorted file.
Total cost: 610 I/O's.

c. [2pt] What would the standard external sort algorithm cost to sort table **R** in Question 3b, again with an allocation of five (5) buffer frames?

Is Dr. Récorde's version an improvement in this case?

- *Pass 0: 19 times*
  - *reads in 5 pages, sorts across them*
  - *writes a 5-page run*

  Makes runs $r_{0,0}, \ldots, r_{0,18}$.
  Cost: $2 \times 95 = 190$.
- *Pass 1:*
  - *merges runs $r_{0,0}, \ldots, r_{0,15}$ in four 4-way merges of sequential groups of four*
  - *writes out four 20-page runs $r_{1,0}, \ldots, r_{1,4}$*
  - *merges the final three runs $r_{0,16}, \ldots, r_{0,18}$, writing 15-page run $r_{1,5}$*

  Cost: $2 \times 95 = 190$.
- *Pass 2:*
  - *merges runs $r_{1,0}, \ldots, r_{1,4}$ writing run $r_{2,0}$*
  - *merges run (?) $r_{1,5}$ writing $r_{2,1}$*

  Cost: $2 \times 95 = 190$.
- *Pass 3:*
  - *merges the two runs $r_{2,0}$ and $r_{2,1}$ writing run $r_{3,0}$*

  Cost: $2 \times 95 = 190$.

Run $r_{3,0}$ is the sorted file.
Total cost: $2 \times 4 \times 95 = 760$ *I/O's (four passes).*
Note that an optimization the regular algorithm can *make is not to do a merge of* one *remaining run, like in Pass 2 here, since this does nothing. That would save* $2 \times 15 = 30$ *I/O's here, reducing the total to 730 I/O's. In any case, Dr. Recordé's algorithm did better in this example.*

4. [10pt] **Query Plans.** *Time to move on to Plan B.* Short Answer

For Question 4a.

Schema:

Student(<u>sid</u>, sname, startdate, major, advisor)
  FK (advisor) refs **Prof** (pid)
Class(<u>cid</u>, dept, number, section, term, year, room, time, pid, ta)
  FK (pid) refs **Prof**
  FK (ta) refs **Student** (sid)
Enrol(<u>sid</u>, <u>cid</u>, date, grade)
  FK (sid) refs **Student**
  FK (cid) refs **Class**
Prof(<u>pid</u>, pname, pdept, office)

Assume no attribute is nullable. The attribute pid in **Class** refers to the the professor / instructor for the class. The attribute ta in **Class** refers to the teaching assistant for the class. The attribute advisor in **Student** refers to the student's academic advisor.

Statistics:

- **Student**: 50,000 records on 1,000 pages
  - advisor: 2,500 distinct values
- **Enrol**: 2,000,000 records on 20,000 pages
  - sid: 50,000 distinct values
  - cid: 80,000 distinct values
- **Class**: 80,000 records on 1,600 pages
  - pid: 4,000 distinct values
  - ta: 5,000 distinct values
- **Prof**: 4,000 records on 40 pages

Indexes:

- **Student**:
  - clustered tree index on sid (200 data entries per page)
- **Enrol**:
  - clustered tree index on cid, sid (167 data entries per page)
  - unclustered tree index on sid, cid (167 data entries per page)
- **Class**:
  - clustered tree index on cid (200 data entries per page)
- **Prof**:
  - clustered tree index on pid (200 data entries per page)

All indexes are *indirect*, with the records elsewhere. For each tree index, the index pages are 3 deep, except for the index on **Prof**.pid which is 2 deep.

Query:

```
select sid, sname, dept, number, section, term, year, pid
from Student S, Enrol E, Class C
where S.sid = E.sid and E.cid = C.cid
  and S.advisor = C.pid;
```

a. [4pt] Estimate the number of rows the query returns.
Show the steps involved in the estimation.

$$\frac{2M\ (|Enrol|)}{max(2.5K, 4K)} = \frac{2M}{4K} = 500$$

Extra Space

b. [2pt] Why do we use the terminology *outer* and *inner* to refer to the two relations to be joined by a given join algorithm?

This comes from the nested-loop construct:

```
FOR EACH r in R
    FOR EACH s in S
        ...
```

So, we refer to **R** *as the outer, since it is iterated in the outer loop; and* **S** *as the inner since it is iterated in the inner loop.*

c. [2pt] For a multi-relation SQL query, why is the order of joins important?
What can be the difference between different join orders?

The cardinalities of intermediate results can vary greatly.

d. [2pt] What are two advantages that left-deep trees have for query plans?

- *The inners are always base tables.*
  - *This means we have statistics about them (e.g., #rows, etc.)*
  - *and we may have indexes on them.*
- *This increases the chance of* pipelining *operations.*

5. [10pt] **General.** *Psst! They're all 'C'!* MULTIPLE CHOICE

Each is worth one point.

---

a. *Hash join* (which is two pass)
  - **A.** is pipelineable.
  - **B.** preserves the sorted order (if any) from its outer.
  - [✓ **C.**] can be used in some cases when two-pass sort-merge join cannot be.
  - **D.** is always less expensive than an index-nested loop join.
  - **E.** can only be used if the inner has an appropriate hash index on the join-condition attributes.

---

b. *Index-nested loop join*
  - **A.** will often be the best choice when the outer is very large.
  - **B.** does not preserve the sorted order (if any) from its outer.
  - **C.** requires an appropriate index on the join-condition attributes on the outer.
  - [✓ **D.**] requires an appropriate index on the join-condition attributes on the inner.
  - **E.** is not pipelineable.

---

c. Two-pass sort-merge join
  - [✓ **A.**] saves I/O cost over (general) *merge join* when the outer and inner must be sorted explicitly by the query plan.
  - **B.** is pipelineable.
  - **C.** preserves the sorted order (if any) from its outer.
  - **D.** is always preferable to hash join.
  - **E.** requires an appropriate index on the join-condition attributes on the inner.

For Questions 5d to 5f.

The following information is available on tables **Sailors** and **Reserves**.

- **Reserves**: 10,000 records
- **Reserves**.bid: 50 values (1..50)
- **Sailors**: 1000 records
- **Sailors**.level: 10 values (1..10)

The primary key of **Sailors** is sid; of **Reserves** is sid + bid + day. Table **Reserves** reserves has a foreign key on sid referencing **Sailors** (on sid). All columns are *not null*.

---

d.
```
select S.sid, S.name, R.day
    from Sailor S, Reserves R
    where S.sid = R.sid and
          R.bid = 1;
```

Estimate the selectivity of the above query as the number of tuples it likely returns.

A. 2
B. 5
C. 20
D. 50
[✓ **E.**] 200
F. 500

---

e.
```
select S.sid, S.name, R.bid, R.day
    from Sailor S, Reserves R
    where S.sid = R.sid and
          R.bid = 3 and S.level between 4 and 7;
```

Estimate the selectivity of the above query as the number of tuples it likely returns.[3]

A. 1
B. 5
C. 20
[✓ **D.**] 80
E. 100
F. 10,000

---

f.
```
select S.sid, S.name, R.bid, R.day
    from Sailor S, Reserves R
    where S.sid = R.sid and
          S.sid = 13 and R.bid between 1 and 25;
```

Estimate the selectivity of the above query as the number of tuples it likely returns.

A. 1
[✓ **B.**] 5
C. 20
D. 80
E. 100
F. 10,000

---

[3] The *between* predicate in SQL is inclusive.

g. External sort reduces I/O costs by
 A. sorting entirely by main memory.
 $\boxed{\textbf{B.}}$ increasing the fan-in for merge passes.
 C. sorting on the disk, never involving main memory.
 D. using very little buffer pool.
 E. using indexes.

---

h. Replacing *quick sort* in the standard external sort algorithm in *pass 0* with *tournament sort* instead
 A. can take better advantage of sequential reads and writes.
 B. eliminates the need for the merge passes.
 C. is preferable because tournament sort is faster, on average, than quick sort.
 D. can accommodate larger buffer-pool allocations to the operation.
 $\boxed{\textbf{E.}}$ can reduce the I/O costs by sometimes reducing the number of passes.

---

i. Restricting focus to left linear join trees is beneficial for all *except* which of the following reasons?
 A. Left linear join trees typically enable pipelining along the outer relations.
 B. The inner relation for every join is a base table, so a more accurate size estimation of the output can be obtained.
 C. The inner relation for every join is a base table, so an index-nested-loops join remains a possibility.
 $\boxed{\textbf{D.}}$ For any query plan based on a join tree that is not left linear, there is guaranteed to be a query plan based on left linear join tree that is less expensive.
 E. This helps prune the search search space of all possible join trees dramatically.

---

j. The standard approach for cost-based query optimization in relational database systems such as System R is based on
 A. random plan selection.
 B. a greedy algorithm.
 $\boxed{\textbf{C.}}$ dynamic programming.
 D. simulated annealing.
 E. exhaustive search.

Extra Space

You reached the end. Turn in your test. Return to the wild.